Towards a Generic API for Data Load Balancing in Structured P2P Systems

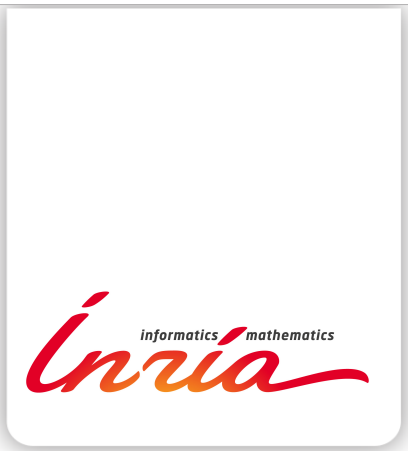
Maeva Antoine, Laurent Pellegrino, Fabrice Huet, Françoise Baude

Towards a Generic API for Data Load Balancing in Structured P2P Systems

Maeva Antoine, Laurent Pellegrino,
Fabrice Huet, Françoise Baude

RESEARCH REPORT
N° 8564
July 2014
Project-Team SCALE

ISSN 0249-6399 ISRN INRIA/RR--8564--FR+ENG



Towards a Generic API for Data Load Balancing in Structured P2P Systems

Maeva Antoine, Laurent Pellegrino,
Fabrice Huet, Françoise Baude

Project-Team SCALE

Research Report n° 8564 — July 2014 — 18 pages

Abstract: Many structured Peer-to-Peer systems for data management face the problem of load imbalance. To address this issue, there exist almost as many load balancing strategies as there are different systems. Besides, the proposed solutions are often coupled to their own API, making it difficult to port a scheme from a system to another. In this report, we show that many load balancing schemes are comprised of the same basic elements, and only the implementation and interconnection of these elements vary. Based on this observation, we describe the concepts behind the building of a common API to implement any load balancing strategy independent from the rest of the code. We then show how this API is compatible with famous existing systems and their load balancing scheme. Implemented on our own distributed storage system, this API integrates well with the existing system and has a minimal impact on its business code. Moreover, this can allow changing only a part of a strategy without modifying its other components.

Key-words: API, Load Balancing, Modularity, Structured P2P

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Vers une API Générique pour Balancer des Données dans les Réseaux P2P Structurés

Résumé : De nombreux systèmes Pair-à-Pair structurés ciblant la gestion de données sont confrontés au problème du déséquilibre de charge. Pour résoudre ce problème, il existe presque autant de stratégies que de systèmes différents. En outre, les solutions proposées sont souvent couplées à une API qui leur est propre, ce qui rend difficile le portage d'une solution de répartition de charge d'un système à un autre. Dans ce rapport, nous montrons que de nombreux dispositifs d'équilibrage de charge sont constitués des mêmes éléments de base, et que seules la mise en œuvre et l'interconnexion de ces éléments varient. Partant de ce constat, nous décrivons les concepts derrière la construction d'une API générique pour mettre en œuvre une stratégie d'équilibrage de charge qui est indépendante du reste du code. Nous montrons ensuite comment cette API est compatible avec certains systèmes existants et leur solution d'équilibrage de charge système. Mise en place sur notre propre système de stockage distribué, l'API proposée s'intègre bien et a un impact minimal sur le code métier. En outre, notre solution permet de changer une partie d'une stratégie sans modifier ses autres composants.

Mots-clés : API, Répartition de Charge, Modularité, Réseaux P2P Structurés

1 Introduction

Load balancing is at the heart of many Peer-to-Peer (P2P) works in particular, systems geared towards data storage and retrieval, to address performance issues that stem from load imbalance on peers. Imbalances may be caused by an unfair partitioning of network identifiers between peers, frequent arrival and departure of peers but also the heterogeneity in terms of bandwidth, storage and processing capacity between machines where peers are deployed. Other reasons can be related to the variation of size, popularity and lexicographic similarities among resources handled by P2P networks. The works considering this area of research can be classified into two main groups: *static* or *dynamic*. In the former, the system load is assumed stable. Either no continuous insertions or deletions are performed and queries remain similar or solutions are based on a fixed and preconfigured set of rules. Furthermore, churn is often evicted and the load balancing decision is assumed to be taken during the join of a peer. The latter enables decisions and adaptations at runtime while usually taking into account endless data insertions and removals but also turnover among peers, namely arrival and departure.

In structured P2P networks, peers manage part of a common identifier space, which can be a circle segment [1], an hypercube subset [2] or a subtree [3]. Usually, resources or data that have to be indexed into the network are assigned an identifier from the common identifier space. This enables routing based on the range a peer is responsible for. This identifier can be the data itself or a hash value associated to the information when a Distributed Hash Table (DHT) or consistent hashing are at the basis of the indexing scheme. Eventually, the information is indexed on the peer managing the resource identifier.

To address load imbalance issues in structured P2P networks, especially regarding data distribution, several load balancing strategies have been proposed based on replication or relocation. The model followed by load balancing strategies usually consists in controlling resources location, peers location or both. However, many variants are conceivable based on indirection, identifiers or range space reassignation and virtual peers[1]. Moreover, designing a load balancing solution requires to consider additional parameters such as the overload criteria to take into account, how overload is detected, and how load information is exchanged. This variety of parameters has led to the definition of multiple solutions that often differ by minor but subtle changes.

We propose to describe the main concepts behind an API for load balancing in structured P2P systems. More precisely, we will focus on the context of data management systems. Indeed, this work was motivated by the building of a distributed platform for data storage and retrieval (see Section 5 for more details). However, the general ideas presented in this paper can be applied on other types of distributed systems. To this aim, we provide a guide of what criteria are important to define and the essential principles to think about before implementing a load balancing strategy. We propose to decompose into components the main features arising from a load balancing mechanism. This enables changing only a part of a strategy without having to impact the other components. When conceiving a distributed system, it is often not so easy to anticipate which kind of load balancing strategy will be the most efficient. Our contribution is to isolate what concerns load balancing from the rest of the code. Separation of concerns prevents from changing code that is not necessary to modify, and provides reusability and better understanding of the code. Therefore, our solution is especially useful when it becomes necessary to experiment with (prior to a definitive choice for instance) various load balancing strategies on a system, without impacting the existing business code.

The rest of the paper is structured as follows. Section 2 introduces some existing load bal-

[1] Unlike traditional P2P networks where one peer is deployed per node, virtual peers are an abstraction allowing several peers to be hosted on a same physical node. Upon the detection of an underloaded or overloaded peer, virtual peers are reassigned to other nodes in order to maintain the machine load under a given threshold.

ancing solutions in the literature we find relevant in our context. Section 3 describes what we consider as the main elements that make up a load balancing strategy. Section 4 presents our common API to implement any kind of strategy. Section 5 describes the experiments and presents the results obtained to balance the load of our own distributed storage system. Section 6 shows how our API can be applied to the previously mentioned papers. Finally, Section 7 concludes the paper.

2 Existing Systems

Many papers propose load balancing solutions for distributed systems, using various strategies. In the following, we focus on four different systems, each implementing its own load balancing strategy. Although the chosen papers do not constitute an exhaustive list of load balancing solutions, they are representative of existing works. Indeed, these strategies are applied on various P2P systems (CAN, Chord) and in different contexts (publish/subscribe, distributed games, data storage). Load balancing is triggered at different states: when a new peer joins the system, when inserting data, or after a peer has periodically compared its load state with an internal threshold or load information received from other peers. Besides, these papers are among the most-cited for the topic.

2.1 Rao et al.

Rao et al. [4] suggest three different strategies based on virtual peers to address the issue of load imbalance in P2P systems that provide a DHT abstraction. This paper proposes a general solution, not especially dedicated to data load balancing. Each physical node is responsible for one or more virtual servers, whose load is bounded by a predefined threshold. A node is considered as imbalanced depending on this threshold: heavy if its load is above, light otherwise. The proposed load balancing solutions are meant to transfer the load between heavily and lightly loaded nodes by moving virtual peers only. The first scheme called one-to-one involves two peers to decide whether a load transfer should be performed or not. A peer simply contacts a randomly chosen peer, and both exchange their load information. If one of them is heavy and the other one is light, then a virtual server transfer is initiated. The second scheme relies on directories indexed on top of the existing overlay. Each directory, indexed on a node, stores piggybacked load information from light nodes. When a node receives a message from a heavily loaded node, it looks at the light nodes in its directory in order to transfer the heaviest virtual server from the heavily loaded node to a lightly loaded one. Finally, the third variant extends the first two by matching many heavily loaded nodes to many lightly loaded nodes, still using directories. A node responsible for a directory receives load information from both heavy and light nodes. This node periodically performs an algorithm to calculate how to balance the load between all these nodes. Solutions specifying which virtual servers should be transferred to which nodes are then sent to the concerned nodes.

2.2 Gupta et al.

Gupta et al. [5] exploit the characteristics of CAN and their publish/subscribe system (Meghdoot) properties to balance the load when new peers are admitted into the system. Each peer in the system periodically propagates its load to its neighbors. When a new peer wants to join the system, it contacts a known peer in the system, responsible for locating the heaviest loaded peer. The authors distinguish subscriptions load from events load given that they have to be handled differently. To balance subscriptions load, the idea is to split a heavy peer's zone so

that its number of subscriptions is evenly divided with the peer that joins. The second solution addresses load imbalance regarding events. If the peer is loaded due to event propagation, alternate propagation paths can be created by using replication. When a new peer p_j joins a peer p_i overloaded by events, the zone from p_i is replicated on p_j (including its subscriptions). In addition, the neighbors are updated to keep track of p_j in a replica list. Finally, events are balanced during the propagation of an event to be matched with candidate subscriptions by picking, on the peer that executes the routing decision, one replica peer out of the list of replicas in a round robin fashion. This replication strategy improves load balancing, data availability and performance.

2.3 Bharambe et al.

In [6] the authors present Mercury, a system made to support range queries on top of a structured P2P network constructed by using multiple interconnected ring layers where each one is named a hub. Each hub manages the indexation of an attribute from a predefined schema. Mercury does not use hash functions for indexing data and suffers from non-uniform data partitioning among peers as data requires to be assigned continuously for supporting range queries. Owing to this bad data distribution, the authors propose load balancing mechanisms based on low overhead random sampling to create an estimate of the data value and load distribution. Basically, each peer periodically sends a probing request to another peer using random routing. This offers a global system load assessment whose values are collected into histograms maintained on peers. Using this information, a heavily loaded node can contact a lightly loaded node and request it to leave its location in the routing ring and re-join at the location of the heavy node. The authors show this approach is enough for effective load balancing because their system topology is an expander graph with a good expansion rate. In other words, with a small number of edges in their network topology, everyone can reach other edges by many paths.

2.4 Byers et al.

In [7] the authors investigate the direct applicability of the power of two choices paradigm [8] on the Chord P2P network for addressing load imbalances in terms of items per peer. The scheme they applied to balance the load between peers can be summarized in a few lines. A node that wishes to insert an item applies d hash functions on the item key and gets back d identifiers (each hash function is assumed to map items onto a ring identifier). Afterwards, a probing request is sent in parallel for each identifier from the identifiers computed previously and the peers managing the identifiers answer with their load. Once load information is retrieved, the peer with the lowest load is adopted for indexing the item. In addition to storing the item at the least loaded peer p_i, this variant consists of adding a redirection pointer (key space identifier) to p_i on all other peers p_j where $j \neq i$. Thus, a lookup can be achieved by using only one hash function among d at random. Following the same principle, load-stealing and load-shedding strategies can be used, too. An underloaded peer should be able to steal items for which it has a redirection pointer, whereas an overloaded peer can pass on an item by creating a redirection pointer to a lighter peer. The experimental results show that using two hash functions ($d = 2$) is enough to achieve a better load balancing with their two choices strategy rather than using a limited number of virtual peers.

3 Load Balancing Differentiators

Even though they seem very different, all the load balancing strategies cited above and most other existing solutions rely on the same principle. A peer decides to move a given amount of *load* (regardless of the type of *load*: bandwidth, CPU, storage) to a certain *target* which will become responsible for the *load* being moved. The decision to move load always comes after a load comparison with a given *source* of information. It is very common to trigger this load comparison during a specific state of the system such as network construction, data insertion or periodically.

Overall, we identified the following differentiators to establish a load balancing strategy. These differentiators represent main concerns to focus on in order to develop a strategy.

a) Criteria Before fixing load imbalances, disproportion in terms of load must be detected. This implies to know which load criteria are involved and how their variation could be measured on peers. This first differentiator defines which load variations are considered and to which resource(s) (CPU, bandwidth or disk usage) but also operation(s) (e.g. item lookup, item insertion, etc.) they refer. Usually, a few criteria, not to say only one is taken into consideration.

b) Load State Estimation Algorithm Once criteria are defined, the next step consists in deciding whether a peer is experiencing an imbalance or not. The purpose of this differentiator is to define how the decision is made. Usually, a peer may rely on a source of load information containing aggregated remote information (see differentiatior g) to figure out how this source of information is populated with remote information) or use purely local information by comparing its local load(s) with predefined threshold(s).

c) Load Balancing Decision The decision to trigger load balancing often differs from a load balancing strategy to another. This differentiator aims to identify when the decision to evaluate load state is triggered. Consequently, it is related to the time at which the whole load balancing mechanism is triggered and will necessarily impact how a load balancing implementation is welded to an existing system business code.

d) Load Balancing Method The method identifies which well known solution is applied to move load from a peer to its *target*. As summarized in the introduction, it may consist in using *virtual peers*, *redirection pointers* or even *range space reassignation*. It helps checking whether prerequisite abstractions that are required to define a given load balancing strategy are available or not.

e) Load Moving Once an imbalance is detected, the next stage is to fix it. It implies to know what is the load to move. This differentiator defines the amount of load to move from a peer to its *target* but also which part.

f) Target Given a peer p that suffers from load imbalance, its target is a set of peers that is used to balance its load with. In other words, it describes who receives the load when load balancing is triggered.

g) Load Information Exchange Load balancing strategies optionnally embed a mechanism to exchange information. It is often used to compare the local load to an average system load estimated through load estimations that are exchanged. This differentiator defines when load

estimations are transferred (if they are), from who and how. Once exchanged and received on a peer, these estimations compose a *source* of information.

h) Load Information Recipients Given a peer p, recipients are peers that share load information with p. They are mainly used to build a *source* of information involved in the load balancing decision process.

4 Generic API for Load Balancing

Defining a generic load balancing API requires to identify key abstractions suitable to model any strategy. In this section, we classify the differentiators presented in section 3 before identifying those that achieve a common purpose, thus allowing us to define components, their wrapping composites and how they compose. After defining components, we give details about embedded functions required to develop load balancing strategies using generic abstractions. An approach based on hierarchical components was deliberately used because components enable modularity and cohesion [9], which eases reusability.

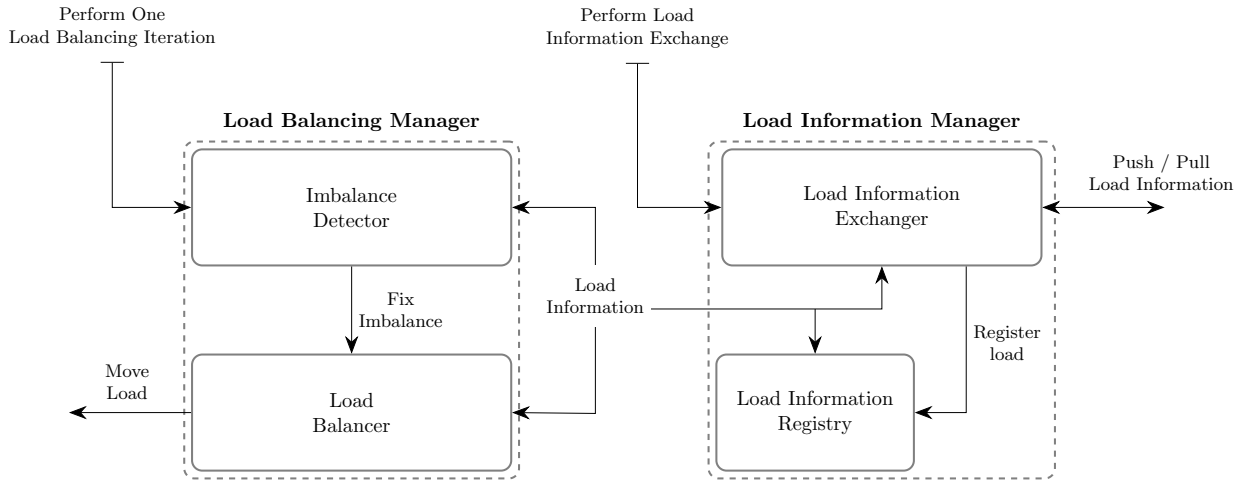


Figure 1: Basic abstractions per peer for a generic API.

4.1 High-level Abstractions

Features associated to differentiators a) to f) relate to the management of load balancing and could be gathered in a so called *Load Balancing Manager* component. By pushing our analysis deeper, we may argue that differentiators b), c) and e) to f) identify two separate subcomponents. Indeed, the first group of differentiators relates to the detection of imbalances and could be named *Imbalance Detector* whereas the second, called *Load Balancer*, captures the method and the information required to balance the load in case of imbalance. Finally, differentiators g) to h) are merely involved in the process to give feedback about resource utilization per criterion to peers. In a component world, this could be modelized as a *Load Information Manager* with a subcomponent, dubbed *Load Information Exchanger*, in charge of exchanging load information.

Figure 1 illustrates these components in charge of isolating load balancing features on each peer. In addition to the components presented above, the figure sketches an additional one,

named *Load Information Registry*, that aims to link the two main composite components (*Load Balancing Manager* and *Load Information Manager*), since each may run in its own flow of control.

Components are wired together by calling actions on other components. Some actions carry *load information* which contain the following values:

- *peer*: the peer sending its *load information*. Can be a peer identifier, a reference, etc.
- *criterion*: type of *load* (disk space, CPU consumption, bandwidth, etc.).
- *load*: load of the *peer* for a given *criterion*.

These attributes and their value can be expressed in the form of a key/value list. Optional elements such as *optimal load* or *internal threshold* can also be included. Details about internal components actions and their behavior are given in the next subsection.

4.2 Core API

Function calls defined below capture the core of load balancing strategies, classified per component. The signature for required functions is given in a simple untyped pseudo language, thus allowing any particular implementation.

Before entering into the description of the API for each simple component, it is worth noting that the two main composite components identified previously, namely the *Load Balancing Manager* and *Load Information Manager*, expose respectively a **perform_one_load_balancing_iteration()** and a **perform_load_information_exchange()** function. These last act as the entry points for peer instances to execute one step of the two complementary composite components and thus orchestrating in which order functions introduced below are run.

Load Information Exchanger

This component is responsible for sending the peer's load information and receiving load information from other peers in the network.

- **exchange_load_information(**
  $recipients$, $load\_information$**)**
  $\rightarrow load\_information$

A peer sends and receives load information from other peers, for a given load *criterion* (storage, CPU, etc.) and a corresponding amount of *load*. The **exchange_load_information** function may return *Load Information* from pull calls that will be directly used by the *Load Balancer* component or stored in the peer's *Load Information Registry* (see details below). Following the same principle, a push call is used when a peer wants to unilaterally notify *recipients* (a given number of peers: neighbors, all peers, a random peer, etc.) about its load state *Load Information*.

Load Information Registry

This registry stores all *Load Information* received by a peer. Optionally, time can be taken into account when storing information.

- **register(**$load\_information$**)**
- **get_load_report(**$criterion$, $peers$**)**
  $\rightarrow load\_information$

The **register** function writes into the registry *Load Information* received by the peer's *Load Information Exchanger*. The **get_load_report** operation provides *Load Information* of a given set of *peers* according to a certain *criterion*. This estimation is calculated thanks to the *Load Information* messages received and stored earlier. The returned *Load Information* can help estimate the overall average load or the load of a given peer, for example. There can be no result if the calling peer has not recently received any *Load Information* message from the concerned *peer(s)*.

Imbalance Detector

Default behavior is to check if a load criterion is unbalanced (overload or underload), in order to trigger a load balancing strategy.

- **make_decision(***criterion***)**
  $\rightarrow$ *load_state*

Using a given algorithm, this function determines whether to induce a load balancing strategy or not, according to a given *criterion*. This operation is basically meant to return an enumerated type: *overloaded* or *underloaded* if a rebalance is necessary, *normal* otherwise. The returned value may depend on a threshold value or not, typically to detect overload or underload. If a threshold value is used, it can be calculated using *Load Information* provided by the *Load Information Manager* (locally, from the *Load Information Registry* using **get_load_report**, or remotely by contacting peers with **exchange_load_information**).

Load Balancer

This component is responsible for balancing the load. The *criterion* and the *load state* selected by the *Imbalance Detector* are used by all methods here. The *Load Balancer* process can be summarized in three steps:

- **select_load_to_move(***load_information_manager*, *criterion*, *load_state*)
  $\rightarrow$ *load_to_move*
- **select_target(***load_information_manager*, *criterion*, *load_state*)
  $\rightarrow$ *target*
- **rebalance(***criterion*, *target*, *load_to_move***)**

The **select_load_to_move** operation is necessary to calculate the amount of *load to move* from one peer to another. Optionally, it is possible to use local or remote information from the *Load Information Manager* to determine how much load has to be moved. The **select_target** function is responsible for finding which peer(s) will receive this *load to move*. To determine this *target*, it is possible to query the *Load Information Manager* (for example, if *target* has to be the least loaded peer known) but it is not mandatory (*target* can be a random peer, a new peer, etc.).

Finally, the **rebalance** method is used to move the *load to move* between the calling peer and the *target*.

5 Evaluation of the Generic API on the EventCloud

This work was originally motivated by the PLAY project[2] which is a platform that allows for "event-driven interaction in large highly distributed and heterogeneous service systems". In this context, we have developed the EventCloud (EC) [10] framework which is a Java software block that offers the possibility for services to communicate in a loosely coupled fashion thanks to the publish/subscribe paradigm but also to store and to retrieve past events in a synchronous manner.

In this section, we assess the proposed API by extending the EC with the load balancing abstractions introduced previously. Before giving details about the load balancing strategies involved with the EC and to explain how components presented in section 4 are implemented, we introduce the underlying EC architecture and why load balancing is considered. Finally, some experiments are reviewed to gauge how flexible the generic API presented in this paper is.

5.1 EventCloud

The EC enables publish/subscribe interactions by means of its event-driven architecture. Subscribers register their interest in some types of events in order to asynchronously receive the ones that are matching their concerns. Events are semantically described as sets of quadruples. Quadruples are in the form of $(graph, subject, predicate, object)$ tuples where each element is named an RDF term in the RDF [11] terminology. The *graph* value identifies the data source; the *subject* of a quadruple denotes the resource that the statement is about; the *predicate* defines a property or a characteristic of the *subject*; finally, the *object* presents the value of the property.

The underlying architecture is based on a slightly modified version of CAN (Content Addressable Network) [2], a structured P2P network which is built on top of a d-dimensional Cartesian coordinate space labeled $\mathcal{D}$. This space is dynamically partitioned among all the peers in the system such that each node is responsible for events in a zone of $\mathcal{D}$. More precisely, an EC is defined with $d = 4$ to map each RDF term of a quadruple to a dimension of the P2P network. The first dimension is associated to the *graph* value, the second dimension to the *subject* value, and so on. Also, in contrary to the default CAN protocol that makes use of hashing to map data onto nodes, the EC uses the lexicographic order to index data. Thus, a quadruple to index is a point in a 4-dimensional coordinate space whereas a subscription, which may be represented as a quadruple with some wildcards such as *S=(g, s, p, ?)*, simply consists in sending S to all the peers that in this example manage the fixed attributes g, s and p on the first three dimensions.

Load balancing with the EC was motivated by the fact that some RDF terms are more popular than others (especially *predicates*) but also because many RDF terms share common prefixes since they are URIs, thus overloading peers by having one or a few adjacent peers from the identifier space managing most data and many indexing no information. The next subsection sketches how an existing system like EC may define, using the API defined in section 4, its own load balancing strategies without difficulty.

5.2 Load Balancing Strategies

Using the proposed API, multiple strategies are conceivable. However, with the EC, the focus is on two key aspects: load criteria and load information exchange.

The former is to provide a load balancing method that allows to consider the unbalance of multiple different criteria. Although the number of items (RDF terms) per peer is critical in the context of the EC, as for other distributed datastores, this is not the only criterion to consider.

[2] `http://www.play-project.eu`

Unbalance may be caused by the execution of multiple different operations that consume different resources but also require distinct mechanisms to fix the imbalance (e.g. subscriptions matching use CPU and replication could be considered to lower the load).

The latter aspect is about information exchanged. The more information is spread, the more precise average system load estimate is. Consequently, a good decision may be taken for load discharge. However, in distributed systems, exchanging information is costly. Thus, depending on the network size, selecting which subset of peers receives load information is crucial. To assess the API, two load dissemination strategies are proposed, namely *absolute* and *relative*. The first aims to detect imbalances without exchanging information between peers, whereas the second relies on information exchanged to detect whether an imbalance is experienced.

In the following, we explain and give details about how components introduced in section 4 are implemented in the EC to manage both aspects.

Load Balancing Manager

The first stage is about the process involved to detect whether a peer is experiencing an imbalance. Since **run_one_iteration()** is the entry point of the *Load Balancing Manager* component, but also because imbalance detection is assumed to be performed periodically in the EC, the management of multiple criteria is made at this level. The content of this function is summed up by Algorithm 1 along with its crucial subcalls.

The entry point function respectively detects (through the **make_decision** function) a peer as imbalanced if its load for a criterion c is respectively k_1 times greater or k_2 times lower than a load estimate e associated to the criterion c that is observed (k_2 must be lower or equal to k_1). The retrieval of a meaningful e value is made possible thanks to information exchanged between peers. Variables k_1 and k_2 are static variables scoped to the lifecycle of the system as for variable C which is a list of criteria defined before the system starts. Order in which criteria are added matters since it defines priorities in which imbalances are detected. Besides, the detection process is sequential for the simple reason that load measurements are not necessarily expressed in the exact same unit but also the fact that actions required to fix imbalances depend on criteria.

Upon imbalance detection (lines 5–8) for a single criterion that relates to disk consumption due to the insertion of RDF data, the **select_target** function is used in order to select a peer from a preallocated pool of peers (line 6). Then, the load to move is selected with **select_load_to_move** (line 7). In the EC, two methods have been implemented to partition the load, one that uses the middle value of the zone a peer is responsible for[3] and one that takes advantage of the centroid value, this last considering the number of items per peer and RDF terms size (i.e. number of characters). Both methods, which may be seen as two different *Load Balancer* component implementations, are used to show below how flexible the API is. Once the target and the load to move are identified, a rebalance is performed (line 8).

Load Information Manager

Load information exchange is an optional process. For this reason and as explained before, two flavors of the *Load Information Exchanger* are proposed: *absolute* and *relative*. With the *absolute* version, threshold values are configured per criterion and passed to peers when they are deployed. These last are upper bound values that allow to signal an overload once they are exceeded. Concretely, defining such a behaviour implies to set variables introduced in Algorithm 1 to specific values. By setting k_1 to 1, k_2 to 0 and e to the desired threshold values, the **make_decision** function works with local knowledge only. The *relative* version requires load information exchange

[3]This is the default CAN rule.

between peers to estimate the average system load that each peer aims to remain close to. Peers in charge of receiving load information registered in the *Load Information Registry* depend on the gossip protocol used. For the following evaluation, a basic strategy that consists in forwarding load information to immediate neighbors is applied.

```
1: function RUN_ONE_ITERATION()
2:     for c ∈ C do
3:         load_state ← MADE_DECISION(c)
4:         if load_state ≠ normal then
5:             lim ← GET_LOAD_INFO_MANAGER()
6:             target ←
                   SELECT_TARGET(lim, c, load_state)
7:             load_to_move ←
                   SELECT_LOAD_TO_MOVE(lim, c, load_state)
8:             REBALANCE(c, target, load_to_move)
9:         end if
10:    end for
11: end function
12:
13: function MAKE_DECISION(c)
14:    m ← GET_LOAD_MEASUREMENT(c)
15:    e ← GET_LOAD_ESTIMATE(c)
16:    k1 ← GET_UPPER_THRESHOLD(c)
17:    k2 ← GET_LOWER_THRESHOLD(c)
18:    if m ⩾ e × k1 then
19:        return Overloaded
20:    end if
21:    if m < e × k2 then
22:        return Underloaded
23:    end if
24:    return Normal
25: end function
```

Algorithm 1 – Load state estimation algorithm.

Regarding the registry that is used to store and retrieve load information exchanged with the *relative* version, it makes use internally of a max-heap like datastructure that ensures retrieval of max items (in terms of freshness) in constant time and garbage collection of outdated information in $\mathrm{O}(\log n)$, where n is the number of load information in the data structure.

In summary, while remaining simple, this load balancing strategy investigates different methods involved in a standard load balancing workflow and easily supports the definition of multiple independent criteria. For this reason, we propose to evaluate it in the next section to prove that it fits with the proposed API.

5.3 Results

The presented strategies have been implemented and assessed with micro benchmarks using real data extracted from a Twitter data flow and up to 32 peers deployed on the French Grid'5000 testbed [12]. The workload is about 10^5 quadruples.

Before evaluating the absolute and relative load information exchange strategies, we have performed an experiment to see what could be the best distribution. The scheme consists in injecting the workload on a single peer and once all quadruples have been stored to start load balancing iterations. Each load balancing iteration consists in picking a new peer from a pre-allocated pool of peers to make it join the most loaded one in the network, thus simulating an oracle. The action is repeated until having a network containing 32 peers. To show the interest of using the centroid, the experiments have been performed, as depicted on Figure 2, by using zones cutting based on their middle or centroid values recorded on the fly. By cutting zones at their middle, the workload is distributed on 4 peers only. However, the same experiment using centroid values distributes the load on all peers with almost two-thirds having their load close to the ideal distribution. Although the distribution is not perfect, it is greatly improved.

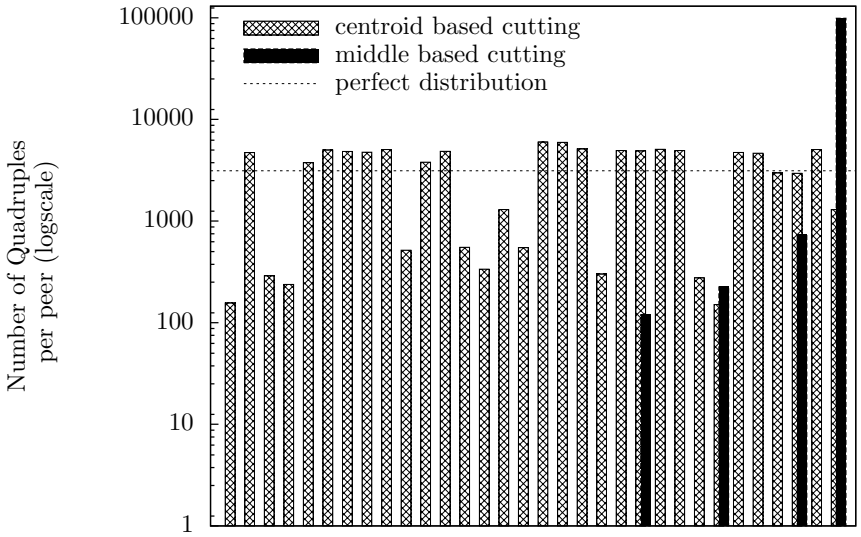


Figure 2: Load balancing partitioning using middle vs centroid.

To compare results for a same configuration (i.e. same workload and number of peers), a good estimator is the coefficient of variation, also known as the relative standard deviation. It is expressed as a percentage by dividing the standard deviation by the mean times 100. In the following we use this estimator to compare strategies. For information, the coefficients are 559.4% and 69.5% when the middle and centroid methodologies are respectively applied to the load balancing experiments presented above, thus showing that centroid performs better because its value is eight times lower than the middle value.

Finally, we have compared the *absolute* and *relative* schemes. For the absolute one, the threshold value is set to the number of quadruples divided by the final number of peers, which gives 3125. The relative strategy does not rely on global knowledge, and k_1 was set to 1.1, so that overload is detected when local measurements on peers are greater than or equal to 1.1 times the estimate value computed by receiving load information from immediate neighbors. The parameter k_1 was set according to emperical evaluations that let suppose the best distribution is achieved for this value.

Table 1 shows the results obtained according to the strategy applied, when correlated with

the results obtained for the previous experiments that exhibit the best distribution that can be achieved (69.5%) due to the "oracle" assumption. The relative standard deviation is almost twice as large (119.75%) as the best when the absolute strategy is applied. Similarly, the relative strategy performs worse than the static load balancing solution but achieves a better distribution (96.57%) than the absolute strategy and this without using global knowledge. Besides, since more peers are receiving RDF data, more nodes are involved to answer subscriptions with the pub/sub layer, thus increasing the throughput in terms of notifications received per second for end-users of EC.

Load Information Exchange Schemes	
Absolute	*Relative*
119.75%	96.57%

Table 1: Load information exchange methods comparison based on relative standard deviation.

Although the analysis of the results is not the central point of this paper, it shows that investigating different implementations for the functions identified in section 4 may have strong impact on results. Thanks to the proposed API, the behaviour of the different load balancing stages can be simply changed by writing a new function with less than 10 lines of code in our case. The main reason is that key features of the load balancing workflow are clearly identified. Obviously, the example shown in this section still requires one line of code change and a full code recompilation to switch from a component implementation to another. In our case, an alternative based on dynamic class loading could be used [13]. In this situation, some code redeployment is required. Moreover, synchronization between nodes may have to be taken into consideration to prevent inconsistent states due to stale information that could transit during the transition from a component implementation to another on peers.

6 API Implementation on Other Systems

As mentioned earlier, this work was motivated by the building of our own distributed storage system (the EventCloud), for which we wanted to apply the most suitable load balancing strategy. Afterwards, we picked various relevant papers from the literature to see if they could validate our generic API, too. This section sketches how these existing load balancing strategies can be implemented using the proposed API. Using our differentiators, presented in Section 3, we were able to decompose in Table 2 each strategy implemented by the papers mentioned in Section 2. Results show that these load balancing strategies can fit our model, even though they seem very different at first sight. As they match our differentiators, we will describe how these strategies integrate with our API.

	Criteria	Load State Estimation Algorithm	Load Balancing Decision	Load Balancing Method	Load Moved	Target	Load Information Exchange	Load Information Recipients
Rao et al.	Resource agnostic (storage, bandwidth or CPU) but only one	Given L_i the load of node i (sum of the loads of all virtual servers of node i) and T_i a target load chosen beforehand, a node is heavily loaded if $L_i > T_i$, lightly loaded otherwise.	Periodically, on each peer	Virtual peers (with no virtual peer split or merge)	One of the overloaded peer's virtual server	A random peer, an underloaded peer or the best underloaded peer according to the scheme used	Random probing for the first scheme (*pull*). Periodic load advertisement from lightly loaded peers (*push*) and sampling from heavily loaded peers (*pull*) with the second scheme. Third scheme implies load exchange from a peer to a directory (*push*)	A peer managing a random id for the first scheme. Directory associated to some peers for the second and third scheme
Gupta et al.	Subscription and data popularity	Always triggering rebalancing when a new peer joins the system	When a peer joins the system	Range space reassignation or replication	Half of a heavy peer's subscriptions or a heavily loaded peer's replica	The heaviest peer in the system known by the new peer joining the system	Periodically, peers update neighbors about their load (*push*) and share their list that contains the k most heavily loaded peers detected	One hop neighbors
Bharambe et al.	Query selectivity and data popularity (routing load)	Lightly loaded if the ratio between its local load (average load of itself, its successor and its predecessor) and the average system load is less than $1 \div \alpha$ but heavily loaded if the ratio is greater than α, with $\alpha \geq \sqrt{2}$	Periodically, on each peer	Range (identifier) space reassignation	Half of the overloaded peer's load	A lightly loaded peer	Periodic random sampling through messages routed with a TTL (*pull*) + each peer periodically samples its one-hop neighborhood to produce a system load estimate (*pull*)	k random peers at TTL hops from the initiator (p) + all one-hop p's neighbors
Byers et al.	Number of data items per peer	Always triggering rebalancing when receiving an item to insert	Upon the insertion of an item (data) on the entity that performs the insertion	Power of two choices paradigm	The item to be inserted	The least loaded peer among those contacted in source for a given item	The peer that wants to insert an item compute d hash values and contact the associated peer to retrieve their load (*pull*)	For d hash functions applied on an item to insert, the n peers managing the computed hash values

Table 2: Load balancing strategies mapped to differentiators.

6.1 Rao et al.

In this paper, peers periodically push their *load information* (*Load Information Exchanger*) to a set of nodes in the system, some of which maintaining a directory (*Load Information Registry*). *Load information* contains the load of each virtual server of a peer and the peer's internal *threshold*. Each peer p periodically compares its load $load_p$ for a given load criteria to its $threshold_p$ (*Imbalance Detector*). Depending on the peer's load state, the paper proposes three different rebalancing strategies (*Load Balancer*):

1. If $load_p < threshold_p$, p is considered as underloaded and a rebalancing is triggered by p. A random node is periodically picked (**select_target**) and its load sent to p (*Pull Load Information* via *Load Information Exchanger*). If the random node is heavy, then a virtual server transfer (**select_load_to_move**) may take place between the two nodes (**rebalance**).

2. If $load_p > threshold_p$, p is overloaded and contacts one of the peers holding a directory to request a lightly loaded peer (**select_target**) and which virtual server (**select_load_to_move**) should be moved (*Pull Load Information* via *Load Information Exchanger*). After information is received by p, the **rebalance** method can be called.

3. If $load_p > threshold_p$, p can also send its load information to a peer *dir* holding a directory (*Push Load Information* via *Load Information Exchanger*). After *dir* has received enough information from heavy and light nodes, *dir* will perform an algorithm to pick which virtual server p should send (**select_load_to_move**) to which light node (**select_target**). The solution will then be sent back to p, to start the **rebalance** process.

The workflow associated to the second strategy is depicted in Figure 3. Steps are numbered to sketch the sequence of actions involved in a typical load balancing iteration with three peers. Arrows between function calls depict remote communications.

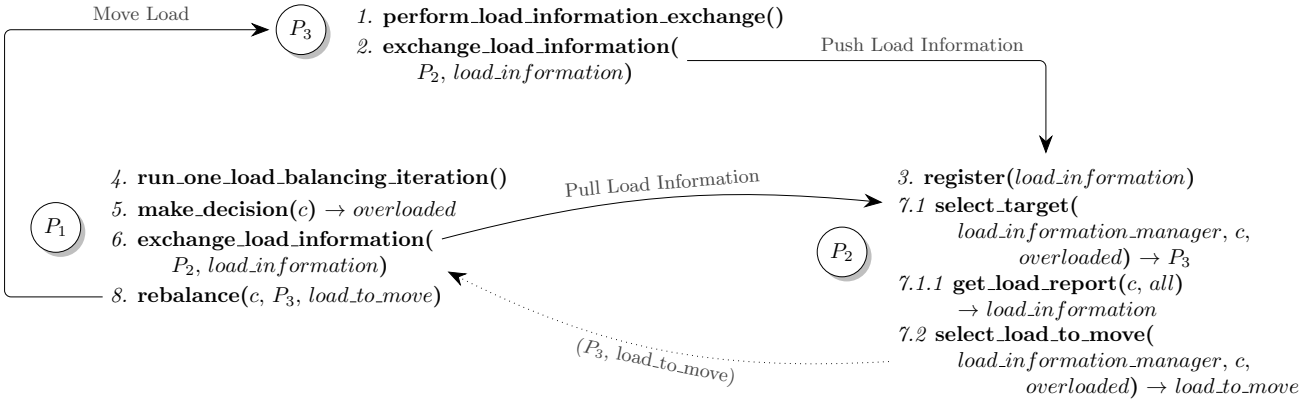


Figure 3: Workflow associated to the second scheme proposed by *Rao et al.*

6.2 Gupta et al.

Peers periodically exchange their load information with their neighbors (*Load Information Exchanger*), as well as an estimated list containing the most heavily loaded peers they know (from

their *Load Information Registry*). Load balancing is only triggered (*Imbalance Detector*) when a new peer p wants to join the system. The first step of the rebalancing process (*Load Balancer*) is for p to find an overloaded peer in the system (**select_target**). To do so, p sends a *pull request* to a random peer. Then, the random peer will look at its registry in order to tell p which node (*target*) is the most overloaded to its knowledge. Finally, *target* is contacted by p. If the overload is due to high storage load, p will split with *target* and receive half of *target*'s subscriptions. Otherwise, if *target* is overloaded because of its processing load, p will replicate *target*'s events and subscriptions (**select_load_to_move** and **rebalance**).

6.3 Bharambe et al.

Peers periodically exchange their load information (using *Load Information Exchanger*). They also maintain histograms containing information about which parts of the network are lightly loaded (*Load Information Registry*) regarding data popularity. Using information contained in its registry, a peer can decide whether it is overloaded or not (*Imbalance Detector*). If so (*Load Balancer* process), the peer contacts an underloaded peer (**select_target**) from a lightly loaded zone in the overlay (thanks to local information collected in its registry). Then, it requests this lightly loaded node (*target*) to leave its position and re-join at the location of the overloaded peer. Finally, the overloaded peer can send part of its data (**select_load_to_move**) to its new neighbor (**rebalance**).

6.4 Byers et al.

A peer having to insert a data item into the system triggers the process (*Imbalance Detector*). This peer applies n hash functions on this item. Then, it contacts each peer associated to an hash function (*Load Information Exchanger*) to pull load information concerning the amount of items already stored by each of these peers. The *Load Balancer* then selects the lightest peer (**select_target**) and sends it the item to be inserted (**rebalance**).

7 Conclusion

In this paper, we have described concepts behind the building of an API for load balancing in structured P2P systems. Many papers propose different load balancing strategies. We have presented four different schemes from some of the most-cited papers for the topic. Their strategies are triggered at various moments (new peer joining the system, data insertion, periodically), impact more or less peers and require to move or replicate data. By decomposing a strategy into essential differentiators, we have shown it is possible to implement these different solutions using our generic API. Regarding the programming aspect, the API allows to separate the code concerning load balancing from the rest of the system. To further assess its utility, the API has been used to evaluate different load balancing methods with an existing system named EventCloud.

Although we presented this API in the context of structured P2P systems, the ideas introduced in this paper could most probably be applied to other networks based on unstructured overlays, or even on a client/server architecture. Finally, modeling such a process can lead to implement adaptive load balancing strategies, for example, by changing one of the key features in a load balancing workflow at run-time. This last is an interesting perspective.

Acknowledgments

This work was in part supported by the EU FP7 STREP project PLAY. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed (see `https://www.grid5000.fr`).

References

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, 2001.

[2] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*. ACM, 2001, vol. 31, no. 4.

[3] K. Aberer, "P-grid: A self-organizing access structure for p2p information systems," in *Cooperative Information Systems*. Springer, 2001.

[4] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in structured P2P systems," in *Peer-to-Peer Systems II*. Springer, 2003, pp. 68–79.

[5] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi, "Meghdoot: content-based publish/subscribe over P2P networks," in *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*. Springer-Verlag New York, Inc., 2004, pp. 254–273.

[6] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 353–366, 2004.

[7] J. Byers, J. Considine, and M. Mitzenmacher, "Simple load balancing for distributed hash tables," in *Peer-to-peer Systems II*. Springer, 2003, pp. 80–87.

[8] M. Mitzenmacher, "The power of two choices in randomized load balancing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 10, pp. 1094–1104, 2001.

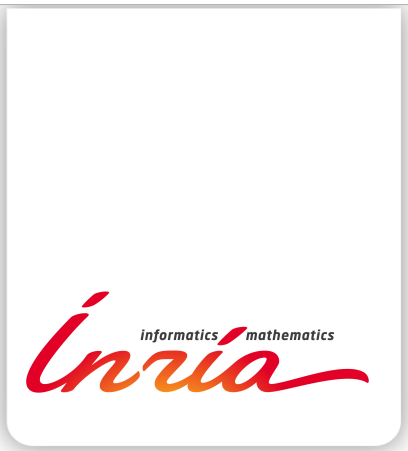
[9] G. T. Heineman and W. T. Councill, "Component-based software engineering," *Putting the Pieces Together, Addison-Westley*, 2001.

[10] L. Pellegrino, "Pushing dynamic and ubiquitous event-based interaction in the Internet of services: a middleware for event clouds," PhD Thesis, University of Nice Sophia Antipolis, Apr. 2014. [Online]. Available: http://tel.archives-ouvertes.fr/tel-00984262

[11] O. Lassila and R. R. Swick, "Resource description framework (RDF) model and syntax specification," 1999.

[12] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jégou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab *et al.*, "Grid'5000: A large scale and highly reconfigurable grid experimental testbed," in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2005, pp. 99–106.

[13] S. Liang and G. Bracha, "Dynamic class loading in the Java virtual machine," *ACM SIGPLAN Notices*, vol. 33, no. 10, pp. 36–44, 1998.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399